"Planning and Heuristic Search"
Oral Pre-Exam Questions (WS 2018/19)

Disclaimer: the given answers are not official.

1 What are nodes and edges representing in an OR graph?

Nodes represent states of a given problem (e.g. board configurations in the 8-puzzle). Edges represent operations (solution steps) that should simplify the problem.

2 What is a solution path in an OR-graph?

A path P in a state-space graph G from node n to goal node γ in G, satisfying given solution constraint, is called a *solution path* for n.

What is a solution base?

A path P in G from n to some node n' is called *solution base* for n.

3 What are constraint satisfaction problems? What are optimization problems?

Constraint satisfaction problems are problems where a solution has to fulfill certain constraints and shall be found with minimum search effort. Optimization problems have to find a solution that in addition to the satisfication of constraints also has to stand out amongst all other solutions with respect to a special property.

4 What is an appropriate representation for infinite graphs?

The implicit representation is appropriate for infinite graphs as it uses the computable methods *successors()* or *next_successor()* to determine the direct successors of a node. Its counterpart the explicit representation can only handle finite graphs, as the graph G = (V, E) is explicitly definied.

5 What is node expansion?

Applying the function *successors(n)* on a node n and thereby creating all direct successors of this node in **one time step** is called *node expansion*. (All algorithms we considered use node expansion as a basic step, except for the backtracking algorithm)

What is node generation?

Applying the function *next_successor(n)* on a node n and thereby creating an unseen direct successor (one at a time) is called *node generation*.

What are the states of nodes?

- generated (living on OPEN)
- explored (neither on OPEN or CLOSED in A*, since it means *next_succesor* was applied and there is still at least one unseen node which will be returned by the next call of *next_succesor*.)
- expanded (living on CLOSED, except for reopening in A*)
- unseen

6 What are locally finite graphs? Why do we need them?

Local finitness means that a node has only a finite number of direct succesors. We do need this property as otherwise we would be stuck in an infinite loop when calling the methods *succesors* or *next_succesor*.

7 What is a solution base (differences to solution paths)?

See question 2. Informally it is the inital part of a possible solution path. Note that we don't necessarily need to find a goal node when following expanding a solution base.

8 What is an efficient way of representing solution bases?

Solution bases can be efficiently represented by a **backpointer path**. A *backpointer* is a reference of a newly generated node, pointing to its parent.

9 What is the tree Basic-OR-Search maintains?

The tree maintained by Basic-OR-Search is called a *traversal tree*. It is rooted in the start node s and definied by node instances from G and edges reverse to the backpointers. Path $(n, \ldots, s)$, defined by the backpointers of the nodes, is reversed in the traversal tree as $(s, \ldots, n)$ and is called a backpointer path.

10 Why is the graph maintained by Basic-OR-Search a tree?

(A traversal tree is not directly maintained by an algorithm, hence this question is not exact.) It is called traversal tree, because it is a tree-unfolding of a part of the explored subgraph of G. Every instance of a node can only have one parent,as it only got one backpointer reference. We don't have cycles because a node in G is represented by multiple instances in the traversal tree (if the node got multiple successors that is).

11 Why is the traversal tree in Basic-OR-Search no subgraph of the OR-graph?

In general traversal trees are no subgraphs of the search space graph G because traversal trees can contain multiple instances of nodes in G.

12 Are DFS and BFS variants of Basic-OR-Search? Why? / Why not?

Yes. They use the same algorithm but their datastructure is implemented differently. DFS uses a *stack* as OPEN list, hence it uses the LIFO principle. BFS on the other hand uses a *queue* as OPEN list, hence it uses the FIFO principle. Both algorithems can be converted into each other by only changing the implementation of the data structure of the OPEN list. ...

13 Comparison of DFS and BFS: Which algorithm is to be preferred when and why?

DFS is preferred when:

- We are given plenty of equivalent solutions.
- Dead ends can be recognized early, i.e. with considerable look-ahead.
- There are no cyclic or infinite paths (or at least they can be avoided).

BFS is preferred when:

- We know a solution is not far away from the start node.
- Solutions are rare and the tree is deep. (Because DFS will take way longer on those graphs.)

An issue with BFS is that it has to store the explored part of the graph completley in memory. So when a graph is very wide, BFS will need too much memory. However, with BFS we do terminate with a solution if one exists, whereas DFS could follow an endless fruitless path.

14 Which nodes are stored on OPEN, which nodes on CLOSED?

Nodes waiting to be expanded are storted on OPEN (this includes both *generated* and *explored* nodes which have not been expanded yet), whereas already expanded nodes are stored on CLOSED.

15 Why is a function cleanup_closed() needed in DFS?

It is needed in order to not run out of memory because of nodes we actually don't need to reference anymore. The function *cleanup_closed* deletes nodes from the CLOSED list that are no longer required. It is based on the principles that nodes which fulfill the dead end requirement can be discarded without hesitation. If a node n is discarded, check if n has any predecessors that are still part of a solution path. A node is part of a solution path if it has a successors on OPEN. Predecessors that are not part of a solution path can be discarded.

16 What is iterative deepening?

Increasing the depth-bound of DFS in an outerloop and run DFS with an increased depth-bound value k over and over again.

17 What information sources does the evaluation function f in BF use?

1. evaluation of the information given by node n
2. estimates of the complexity of the remaining problem at n in relation to Γ

3. evaluations of the explored part G of the search space graph
4. domain specific problem solving knowledge

18 What are the main differences between UCS and BF?

Cost values in uniform-cost-search (UCS) are stored with the nodes. The OPEN list is organized as a heap, and nodes are explored with respect to the cheapest cost. UCS is also called "cheapest-first-search" and it basically is best-first-search (BFS) with some modifications.

19 What is the difference in the evaluation functions of UCS and BF?

Uniform-cost-search can only make use of the knowledge from the explored part of the search space graph, the evaluation function f in BF can use domain specific knowlege.

20 What is path discarding?

For two paths leading to the same node, the one with the higher f-value is discarded. (Path discarding is only used in BF* and its successors.)

21 When using path discarding, is the traversal tree a subgraph of the search space graph?

This questions aims at the removal of duplicates. With path discarding the traversal tree will represent cycle-free paths in G starting in s. If G is cyclic-free then it is a subgraph.

22 Why can path discarding be problematic?

Path discarding is irreversible. It entails the risk of not finding desired solutions. The risk can be eleminated by restricting the evaluation function to be orver-preserving. Path discarding is performed implicitly by maintaining at most one instantiation per node and, therefore, one backpointer per node.

23 What does node reopening mean?

Reopening means moving a node from CLOSED to OPEN because the node could be reached with a better f-value than the first time it was explored.

(Note: Reopening of nodes can be avoided by using a monotonically increasing function f, $f(n) \leq f(n')$ for successors n' of n.)

24 What is $C_P(s), C^*(s), \hat{C}_P(s), \hat{C}(s)$?

24.1 $C_P(s)$

C_P is a cost function which assigns each solution path P and node n in P a cost value $C_P(n)$. $C_P(s)$ specifies the cost of a solution path P for s:

$$f(\gamma) = C_P(s)$$

with P backpointer path of γ.

24.2 $C^*(s)$

$C^*(s) = C^*$ is the optimum solution cost for s.
In general: $C^*(n) = \inf \{C_P(n) \mid P \text{ is solution path for } n \text{ in } G\}$

24.3 $\hat{C}_P(s)$

$\hat{C}_P(s)$ is the estimated optimum solution cost for s in G based on a solution base P with cost function $C_P(n)$. $\hat{C}_P(s)$ is optimistic iff $\hat{C}_P(n) \leq C^*_P(n)$.

24.4 $\hat{C}(s)$

$\hat{C}(s)$ is the estimated optimum solution cost for s (no solution base provided. so it searche in all solution bases)
Estimated optimum solution cost for node n in G:
$\hat{C}(n) = \inf \{\hat{C}_P(n) \mid P \text{ is solution base in } G\}$
A solution base P for s with $\hat{C}_P(s) = \hat{C}(s)$ is called most promising solution base (for s).

25 How do we define an evaluation function f by a cost function $\hat{C}$?

UNSURE!
We define $f(n)$ as $\hat{C}_P(n)$ with P backpointer path of n: $f(n)$ is a recursive evaluation function.

26 How do we define recursive cost functions?

A cost function C_P for a solution path P is called recursive if for each node n in P holds: $C_P(n) = \begin{cases} F[E(n)] & n \text{ is leaf in } P, \text{ hence } n \text{ is goal node.} \\ F[E(n, C_P(n')] & n \text{ is inner node in } P \text{ and } n' \text{ direct succ. of } n. \end{cases}$

27 How can we define a function $\hat{C}_P(n)$ for estimated solution cost on basis of recursive cost functions?

$$\hat{C}_P(n) = \begin{cases} c(n) & n \text{ is leaf in } P \text{ and } P \text{ is solution path.} \\ h(n) & n \text{ is leaf in } P \text{ but } P \text{ is no solution path } (n \text{ in OPEN}). \\ F[E(n), \hat{C}_P(n')] & n \text{ is inner node in } P \text{ and } n' \text{ direct successor of } n \text{ in } P. \end{cases}$$

$E(n)$ denotes local properties of n and F is the so called cost measure. P denotes a solution base. The estimated value $h(n)$ is used as if it was correct in the computation of predecessors.

28 Why can it be an advantage to use recursive cost functions?

If we use a recursive cost function, we compute tail to front (Face-Value-Principle). This means going all the way down to γ and then up again, combining the costs on the way. We have to comnsider the whole Path P when using a recursive cost function. The advantage is then that we simply have to consider the costs of a single node and it's local properties instead of the whole path at once. It is basically a series of local computations.

29 Can cost functions help to avoid problems in path discarding?

Idea: Order preserving cost function can help to avoid following the wrong way and thus wrongfuly discarding a optimum paths. Cost estimations for alternative solution bases must be independet of their shared continuation (S:III-80). $\hat{C}_P(n)$ must be *order-preserving*.

30 What is the evaluation function used in algorithm A*?

In A* we use $f(n) = g(n) + h(n)$ as evaluation function.

31 What is h and what is g in the evaluation function of algorithm A*?

g is the sum of edge cost values of the backpointer path from a given node (cost of backpointer path of a given node at some point in time.)

h is the estimate of the optimum cost of a solution path from a given node. ($h(\gamma) = 0$.)

32 What is path cost in algorithm A*?

Sum of edge cost values along that path.

33 Is the underlying path cost function $\hat{C}_P(s)$ in A* order preserving? Is this only true if we have negative edge cost values?

Idea: In A*: $f(n) = \hat{C}_{P_{s-n}}(s) = g_{P_{s-n}}(n) + h(n)$. Evaluation functions f that rely on additive cost measures $F[e, c] = e + c$ are oder-preserving. So $f(n)$ is order-preserving. It is not only true for negative edge cost values.

34 Why do we need delayed termination in order to solve optimization problem? (Example?)

Without delayed termination, BF would return as soon as it found a goal node γ. With delayed termination we terminate when we select the most promising soloution base from OPEN which happens to be a solution path. Assume we do not use delayed termination and we reached node n. The only two goal nodes, γ_1, γ_2 are descendents of n. γ_1 is first explored with a path cost of 100. BF would immediatley terminate with γ_1 instead of exploring all possible children of n, which could lead to exploring γ_2 with a much smaller path cost. This is why we need delayed termination. In this case, we would terminate with the cheaper solution path if it happened to be the most promising solution base at this point in time.

35 What is an optimistic evaluation function?

An optimistic evaluation function underestimates cost. In particular the true cost of a solution path $P_{s-\gamma}$ extending a solution base P_{s-n} exceeds its f-value: $C_{P_{s-\gamma}}(s) \geq f(\gamma)$.

36 Why do we need optimistic evaluation functions in order to solve optimization problems? (Example?)

Assume we have a graph with start node s and it's two children a and b. There is a goal node γ_a and γ_b, reachable from a and b respectively. Also assume we use a non-optimistic evaluation function. If the f-value of b would be 100 but the actual cost for $P_{s-\gamma_b}$ would be 1, we would still follow node a if it's f-value was for example 5. We would continue to follow $P_{s-\gamma_a}$ and reach a goal node. The actual cost would then be 5 which is less than the 1 we would've paid if we followed node b. But b was overestimatic the actual cost, so we never considered the actual cheapest path in this graph.

37 What is the motivation for specifying Prop(G) for search space graphs?

Prop(G):

1. Search space graph G is an implicitly definied OR-graph.
2. G has only a single start node s.
3. G is locally finite.
4. For G a set Γ of goal nodes is given.
5. Each path from s to $\gamma \in \Gamma$ in G is a solution path. In A*(γ)= $true$, independent of the backpointer path of γ.
6. Each edge (n, n') in G has non-negative edge-cost $c(n, n')$. The pathcost is the sum of edge cost along that path.
7. G has a positive lower bound δ of edge costs. δ is fixed $c(n, n') \geq \delta > 0$ for all (n, n') in G.
8. For each node n in G there exists an heuristic estimate $h(n)$ for the cost of the cheapest path from n to γ. $h(n) \geq 0$, since $h(\gamma) = 0$) for $\gamma \in \Gamma$.

Motivations for Prop(G):

1. (a) **Implicitly**: Would not be possible to represent infinite graphs with explicit representation.
   (b) **Directed OR-Graph**: We want to use A*, a BF algorithm for OR-graphs.

2. Simplification.

3. Node expansion. ForEach-Loop would not terminate if *successors* would return infinite many nodes.

4. Maintenance of structure.

5. No additional constraints on solution paths.

6. (a) **non-negative cost**: Prune cyclic paths which would corrupt the backpointer structure.

   (b) **sum of edge cost**: Requirement as A* uses additive cost measure. No other calculation is possible.

7. We get rid of infinite graphs which for example will half the edge cost values all the time, starting with $1/2$ and thus creating an infinite path which we will follow if the other option has edge cost value greather than 1.

8. (a) **computable**: We want to implement it.

   (b) $h(n) \geq 0$: as path cost must not be negative, the estimate should also be non-negative. (Worst case $h(n) = 0$: uninformed search.)

   (c) $h(\gamma) = 0$: Simplicity.

38 What is the consequence of a positive lower bound of edge cost values for long paths?

The cost of the path will eventually exceed any given bound if the path is only long enough due to the lower bound of edge cost values δ .

39 Is existence of optimum cost solution paths guaranteed for search space graphs with Prop(G)?

No. However, if there is a solution path, we can argue that there also is an optimum solution path in G. But we can not guarantee that there is any solution path in G at all.

40 What is completeness, what is admissibility for search algorithms?

Completeness: The algorithm terminates with a solution, if one exists.
Admissibility: The algorithm terminates with an optium solution, if a solution exists.

41 What are the main steps in proving completeness of A*?

1. Assume that there is a solution path $P_{s-\gamma}$.
2. Due to the shallowest OPEN node, no termination.
3. Define $M = max(f(n))$ for $n \in P$.
4. $f(n) \leq M$ for all nodes on P.
5. $f(n') \geq M$ not selected before nodes on P.
6. Finite set of paths with cost M starting in s.
7. γ selected

42 Why can't we prove termination of A* on infinite graphs?

Because in the proof of termination of A* we use the fact that the number of cycle-free paths is finite in a finite graph, and therefore it can only exist a finite number of backpointer paths. We can not use this fact for infinite graphs. In other words: A* could follow an infinite path in an infinite graph and never terminate – hence we can not prove termination on infinite graphs.
ATTENTION: We can prove completness for infinite graphs, when we assume that there is a solution path. But to prove termination we are not allowed to assume this.

43 What is a shallowest OPEN node?

The shallowest OPEN node for a path P is the first node we encounter on that path (starting from s) which is on OPEN.

44 How do shallowest OPEN nodes help proving completeness?

Due to the shallowest OPEN nodes, we can argue that A* will not terminate with "fail" since there always is an OPEN node which has to be selected for expansion.

45 What is the additional property of shallowest OPEN nodes on optimum cost paths that is used for proving admissibility of A*?

We use the lemma of *C*-bounded OPEN nodes* which means that on an optimum cost path there has to be a shallowest OPEN node with $f(n) \leq C^*$.

46 What is the statement of the C* bounded OPEN node lemma?

Let G be a search space graph with $Prop(G)$ and let A* use an admissible heuristic function h. For each optimum path $P^*_{s-\gamma} \in \boldsymbol{P^*_{s-\gamma}}$ and at each point in time before A* terminates there is an OPEN node n' on $P^*_{s-\gamma}$ with $f(n) \leq C^*$.

47 What is the definition of an admissible heuristic function?

$h(n) \leq h^*(n)$ for all $n \in G$. (Admissible heuristic functions are thus optimistic estimates of the cheapest solution cost for a node in G.)

48 What is the idea of the proof of the C* bounded OPEN node lemma?

Assumption of opt. sol. path + combination of shallowest open node + optimally reachable node on optimum solution path + admissible heursitic function.

49 What is the statement of the admissibility theorem for A*?

A* is admissible when using an admissible heuristic function h on search space graphs G with $Prop(G)$.

50 If we use a solution path $P_{s-\gamma}$ with cost $C \geq C^*$ instead of an optimum cost solution path, what is the statement we can prove instead of the C^* bounded OPEN node lemma?

... We can not use the 3. step of the proof (optimally reachable) since we are not on an optimum cost solution path anymore ...

51 What necessary and sufficient conditions for node expansion by A* did we consider?

1. necessary: $f(n) \leq C^*$
   sufficient: $f(n) < C^*$.
2. necessary: $\exists C^*$-bounded path P_{s-n}
   sufficient: $\exists$ strictly C^*-bounded path P_{s-n}.
3. necessary: $g^*(n) + h(n) \leq C^*$
   sufficient: $g^*(n) + h(n) < C^*$.

52 What are the nodes considered in necessary and sufficient conditions for node expansion by A*?

1. nodes on OPEN
2. nodes on $C*$-bounded paths
3. ...

53 How can we increase efficiency by applying the necessary condition for node expansion of OPEN nodes by A*?

...

54 How is monotonicity (consistency) for heuristic functions defined?

Consistency: $h(n) \leq k(n, n') + h(n')$
Monotonicity: $h(n) \leq c(n, n') + h(n')$

55 How can monotonicity be proven from consistency? (Proof ideas.)

Monotonicity follows from consistency since consistency states the triangle inequality for any pair of nodes n, n' with n' reachable from n considering cost of the cheapest path. Monotoniciy considers special pairs of nodes $n' \in succ(n)$ and per definition holds: $k(n, n') \leq c(n, n')$.

56 How can consistency be proven from monotonicity? (Proof.)

Let n_1 be reachable from n_0 and let $P = (n_0, n_1, \ldots, n_l)$ be a cheapest path from n_0 to n_l. Using the monotonicity of h it can be proven by induction over the path length that

$$h(n_0) \leq \sum_{i=1}^{l} c(n_{i-1}, n_i) + h(n_l)$$

Since a cheapest path P was considered, we have $k(n_0, n_l) = \sum_{i=1}^{l} c(n_{i-1}, n_i)$ and hence proven consistency: $h(n) \leq k(n, n') + h(n')$.

57 Why is it important to have both, monotonicity and consistency?

Consistency is the more powerful statement, so for further proofs it makes sense to have consistency. However, to check if a heuristic function h is consistent, is way harder than to check if it is monotone, since $k(n, n')$ had to be considered instead of $c(n, n')$.

58 Are monotone heuristic functions admissible? (Proof.)

Yes.
Since *monotonicity* $\Leftrightarrow$ *consistency*, it is enough to show that consistent heuristic functions h are admissible. Proof (Sketch):

1. Let h be consistent, i.e. $h(n) \leq k(n, n') + h(n')$ for all nodes n, n' in G.
2. Consider an arbitrary node n.
3. If no goal node is reachable from n, then $h^*(n) = \infty$ and thus $h(n) \leq h^*(n)$.

4. If some goal node is reachable from n, then there is a goal node γ reachable from n with cheapest cost (Corollary: Solution Existence Entails Optimum).

5. Using $n' = \gamma$, we have $h(n) \leq k(n, \gamma) + h(\gamma)$, thus $h(n) \leq h^*(n) + 0$.

6. Since n is arbitrary chosen, $h(n) \leq h^*(n)$ holds for all nodes. Hence h is admissible.

59 Consider the 8-puzzle. Give an example of a monotone heuristic function.

TODO: Check if this is true.

1. sum of misplaced tiles (Hamming Distance).

2. sum of manhatten distances of misplaced tiles (Manhatten Distance).

60 What is the advantage of using monotone heuristic functions in A*?

When h is monotone, we don't have to re-expand nodes which are already on closed. More formally: An A* algorithm that uses a monotone heuristic function h will expand only nodes to which it has found cheapest paths: $g(n) = g^*(n), \quad \forall n \in \text{CLOSED}$.

61 Give the outline of the proof of the No Reopening Theorem

1. Assume that A* selects a node n for expansion with $g(n) > g^*(n)$.

2. Let P^*_{s-n} be a cheapest path from s to n.

3. If $P^*_{s-n} \bigcap \text{OPEN} = \{n\}$, then all predecessors of n on P^*_{s-n} have been expanded and $g(n) = g^*(n)$, contradicting the assumption (Lemma: Shallowest OPEN node on optimum path).

4. $P^*_{s-n} \bigcup \text{OPEN} \neq \{n\}$, let n' be the shallowest OPEN node on P^*_{s-n}.

5. Using $g(n') = g^*(n')$ and the monotonicity of h we have $f(n') = g(n') + h(n') = g^*(n') + h(n') \leq g^*(n') + k(n', n) + h(n)$.

6. Since $n' \in P^*_{s-n}$, we have $g(n') + k(n', n) = g^*(n)$ and thus $f(n') \leq g^*(n) + h(n)$.

7. According to the assumption $g(n) > g^*(n)$ we have $f(n') < g(n)+h(n)$ and thus $f(n') < f(n)$.

8. A* selects n' for expansion instead of n, contradicting the assumption.

62 Should we always prefer monotone heuristic functions over admissible ones?

Yes? Since monotonicity requires admissibility but admissibility does not guarantee monotonicity. With monotonicty we can reduce the complexity of A* because of the No Reopening Theorem.

63 If we have two heuristic functions, the one more informed than the other on part A of the search space graph and the other way round on part B, which heuristic function should we use in A* search?

We should prefer the heuristic function that is more informed on part B. It is more important to have a good heuristic when near to a goal node than at the very beginning. Example: Distance Uni -¿ City: not important if 2km or 10km. But distance from here to blackboard is more important if it is 10m or 50cm.

64 Why is solving optimization problems with A* search an efficiency nightmare?

The runtime of A* can be exponential, i.e. 2^n. Also if several near-optimum solutions exist, then A* uniformly follows the different paths, spending a lot of time.

65 What is the idea of the weighing approach?

Evaluation function $f = g + h$ consists of a breadth-first-component (nodes close to s are preferred) g, and a depth-first-component h. The general idea of the weighing approach is now to strengthen the depth-first-component to find "some" solution faster whilst guaranteeing that the cost of the foind solution will be near optimum.

66 Why do we expect that the search effort in WA* is less than in A*?

WA* uses $f_\epsilon = g + (1 + \epsilon) \cdot h$. It allows a solution to be error term ϵ worse than an optimal solution. Therefore we expect that the overall search effort for finding a non-optimal solution will be less than in A*.

67 What properties should h have in WA*, what properties should $(1 + \epsilon)h$ have?

h should be an admissible heuristic function. ϵ should be chosen in such a way that $(1 + \epsilon)h$ is not admissible.

68 What is the idea of the A^*_ϵ algorithm?

Use two heuristic functions h and h_F. h is used to form $FOCAL = \{n \in OPEN \mid f(n) \leq (1 + \epsilon) \cdot \min_{n' \in OPEN} f(n')\}$. $h_F(n)$ is used to estimate the computational effort for completing the search from n and thus for the selection of nodes from FOCAL.

69 What properties should h have in A^*_ϵ, what properties should h_F have?

h should be admissible. No restrictions on h_F.

70 Why do we expect that the search effort in A^*_ϵ less than in A*?

Same reason as question 66, we allow a ϵ-percentage deviation from optimum solution cost.

71 What are the differences of WA* and A^*_ϵ to A* in pseudocode?

WA*: We have to rewrite the selection from open in line 4. It should include the $(1 + \epsilon)h$ instead of only $g + h$.

A^*_ϵ : Add a calculation of FOCAL and then select from FOCAL based on h_F.